**Spark Document**

**Important things and Solutions**

**To change the schema of the data loaded from the CSV like when we have all string**

If you see the schema above all the columns are of type String. Now I want to change the column proc\_date to Integer type and cyc\_dt to Date type, I will do the following:

**So add the columns with different name and change the data type by casting as shown:**

df = df.withColumnRenamed("DatatypeCode", "data\_type\_code")

df = df.withColumn("proc\_date\_new", df("proc\_date").cast(IntegerType)).drop("proc\_date")

df = df.withColumn("cyc\_dt\_new", df("cyc\_dt").cast(DateType)).drop("cyc\_dt")

df.printSchema()

RMSE by itself does not tell whether your estimations are good!

Suppose you are estimating the minimum temperature of a place where its variation is 0-5 C. Then if your RMSE is around 1, your estimations are good.

But if you are estimating the mean sea level pressure of the same region expecting to get the RMSE of 1 is something like a kid asking for the moon. In this case, an RMSE of a few hundred is good enough.

Now, to judge whether your SMSE of a few hundred is good or just a value of 1 is good is judged based upon the data mean.

Hence, a term called scatter index (SI) is defined to judge whether RMSE is good or not. SI is RMSE normalised to the measured data mean or SI=RMSE/measured data mean. If SI is less than one, your estimations are acceptable

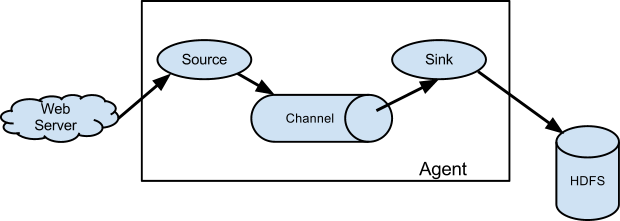
Apache Flume

Apache Flume is a distributed, reliable, and available system for efficiently collecting, aggregating and moving large amounts of log data from many different sources to a centralized data store.

* Source Process bringing the data for aggregation -> Channel(which is memory storage can be even HDFS storage also) -> Sink Process for putting the data from the channel

So till now our py spark was bringing data from the HDFS Sources -> Loading it in memory -> Transforming -> and then storing it back

* But Flume is for aggregating data from various sources and then putting in our HDFS environment so as to then our Pyspark Process can work on it



The use of Apache Flume is not only restricted to log data aggregation. Since data sources are customizable, Flume can be used to transport massive quantities of event data including but not limited to network traffic data, social-media-generated data, email messages and pretty much any data source possible.

( now the above line says that it is not that only that the work of the flume is of the Log aggregation but it can work also on the event based model ----

Event generated 🡪 Flume Agent -> Destination u want can be HDFS

Kafka-> Flume Agent

Flume Agent with Destination to -> Kafka Topics

1. A Flume source consumes events delivered to it by an external source like a web server.
2. The external source sends events to Flume in a format that is recognized by the target Flume source.

For example, an Avro Flume source can be used to receive Avro events from Avro clients or other Flume agents in the flow that send events from an Avro sink. A similar flow can be defined using a Thrift Flume Source to receive events from a Thrift Sink or a Flume Thrift Rpc Client or Thrift clients written in any language generated from the Flume thrift protocol.

1. When a Flume source receives an event, it stores it into one or more channels.
2. The channel is a passive store that keeps the event until it’s consumed by a Flume sink. The file channel is one example – it is backed by the local filesystem.
3. The sink removes the event from the channel and puts it into an external repository like HDFS (via Flume HDFS sink) or forwards it to the Flume source of the next Flume agent (next hop) in the flow.

The source and sink within the given agent run asynchronously with the events staged in the channel

Project Architecture using Flume based

HTTP Source -> Flume Agent-> Spark Sink -> Spark streaming Receiver for processing in micro batches-> Sending results to the RDBMS

Why we use the Kafka/Flume between the Spark streaming

IoT devices are sending tons of data every minute (or seconds to make things worse) and Kafka acts as a **shock absorber** so Spark can process the datasets on its pace.

Spark Streaming (and Structured Streaming) are batch-oriented and pull-based so the data has to be available to be fetched and processed. Kafka (or Cassandra) are often used to take the load and keep the data before Spark is read to handle it. They just make Spark Streaming's life so much easier and peaceful.

**Spark Streaming provides two categories of built-in streaming sources.**

**Basic sources**: Sources directly available in the Streaming Context API. Examples: file systems, and socket connections.

**Advanced sources**: Sources like Kafka, Flume, Kinesis, etc. are available through extra utility classes.

These require linking against extra dependencies as discussed in the linking section.

So the point is that we can pull the data from the basic sources as well as the advanced sources and they act as a shock absorber.

Input D Streams can also be created out of custom data sources. All you have to do is implement a user-defined receiver (see next section to understand what that is) that can receive data from the custom sources and push it into Spark. See the Custom Receiver Guide for details.

When it comes to Streaming data from Flume to Spark you have 2 options.

1. Push Model: Spark listens on particular port for Avro event and flume connects to that port and publishes event
2. Pull Model: You use special Spark Sink in flume that keeps collecting published data and Spark pulls that data at certain frequency

Spark serialization and deserialization

Understanding Spark Serialization , and in the process try to understand when to use lambada function , static, anonymous class and transient references

* Spark follows Java serialization rules, hence no magic is happening.
* Unlike Map Reduce code where you had separate classes for Mapper, Reducer, Driver . In Spark you just have one class which incorporated the logic of the complete ETL(Map, Reduce Task ) and the driver.
* The Spark class is the driver hence all the code you see is executed on driver, hence all object instantiation happens on driver. The serialized objects are sent to Executors to work as Task.
* All Lambda/Anonymous/Static class used with the transformation are instantiated on Driver , serialized and sent to the driver.

Spark code analysis what executes where ( Very important)

**Spark Code Analysis**

Lets take [***WordCount***](http://bytepadding.com/big-data/spark/word-count-in-spark/) example in Spark and lets try to understand whats executed where and how to understand . And also try to understand spark serialization.

In Map Reduce the Driver , Mapper and Reducer code was written as separate class and was very easy to figure out which code is executed where. In spark all the code for driver , Map Task and reduce Task is part of the same class and hence one needs to understand whats going under the hood to understand the same.

( so all tasks are in the same class)

* The spark code follows all the rule of Java , so theres no magic and no changes in the rule.
* The spark code once submitted the framework depending on the Transformation and actions present in the spark code generates a DAG of tasks.
* All transformation Before shuffle are clubbed into the same tasks.
* Operations such as grouby, reduceByKey , combineByKey, AggregateByKey results in a shuffle , in Spark terminology results in stages.
* Job is composed of Stages. the Boundary of a stage is a shuffle operation.

***From Driver Perspective :***

* ToolRunner.run(new WordCount(), args) is executed on the driver.
* The Lambda Function / Anonymous/ Static class used within the Transformation are instantiated on driver.
* The Lambda Function / Anonymous/ Static class used within the Transformations are serialized and sent to the Task as defined by the DAG.
* Actions such as collect(), count() are executed on driver . These actions results in collecting data from tasks(on executors) onto the driver and calculating the sum , average etc. These operations results in lot of data coming on driver.
* Actions like mapPartitions(), map(), reduceByKey(), combineByKey(), aggregateByKey(), groupByKey() are executed on executors as part of Map Task or Reduce Task.

***Executors :***

* Actions like mapPartitions(), map(), reduceByKey(), combineByKey(), aggregateByKey(), groupByKey() are executed on executors as part of Map Task or Reduce Task.
* The function inside these Transformation are received in a serialized from the driver onto the executor .
* All the transformations in the same stage is executed as part of same task in the executor.

|  |  |
| --- | --- |
| **Code** | **Executed on  Driver/Executor** |
| javaSparkContext = getJavaSparkContext | Driver |
| JavaRDD<String> stringJavaRDD = javaSparkContext.textFile(inputPath); | Results in Data Read operation in all  Executors |
| .flatMap | Executor (Map Task) |
| .mapToPair | Executor (Map Task) |
| .reduceByKey | Executor (Reduce Task) |
| .repartition | decides parallelism for Num Reduce Task. (if used after Map task, Results in a Shuffle Task on Executors ) |
| .saveAsTextFile(outputPath); | saves output from all Reduce Task on Executor |
| stringJavaRDD.count() | On Driver , Data Pulled from all tasks to Driver to perform count |

*Resilient Distributed Datasets (RDDs) are a distributed memory abstraction that helps a programmer to perform in-memory computations on large clusters that too in a fault-tolerant manner*